Abstract

Multi-threading is a methodology that has been extremely used. Modern software depends essentially on multi-threading. Operating systems, famous examples, are based on multi-threading; a user can write his document, play an audio file, and downloading a file from internet at the same time. Each of these tasks called a thread. A common problem occurs when implementing multi-threaded programs is a data-race. Data race occurs when two threads try to access a shared variable at the same time without a proper synchronization. A detector is software that determines if the program contains a data-race problem or not. In this paper, we develop a detector that has the form of a type system. We present a type system which discovers the data-race problems. We also prove the soundness of our type system.

Keywords: multi-threaded programs, type systems, data-race, semantics of programming languages

1. Introduction

Developing and debugging software that depends on multi-threading is a tricky mission because of ingrained concurrency and indeterminism. There are many bugs occur according to these properties. Detecting and preventing these bugs are important areas of research. Bugs have several forms. The most extensively studied one is data-race: two concurrent threads accessing the same shared variable without proper synchronization. Data-race detector is a tool that determines whether a program is a data-race free or not. Two approaches are followed when developing detectors: static approach, and dynamic approach. Static detectors determine whether a program produce a data-race regardless of inputs of the program. Apart from static detectors, dynamic detectors determine whether a program produce a data-race of a given inputs at execution of the program.

The advantages of static detectors are the consideration of different execution paths (more elaborate), and the soundness of detector, i.e. proving the bug-freeness of programs. Examples of static detectors (Kahlon, Sinha, Kruus, & Zhang, 2009; Kahlon, Yang, Sankaranarayanan, & Gupta, 2007; Naik, Aiken, & Whaley, 2006; Young, Jhala, & Lerner, 2007). On the other hand, dynamic detectors like (Savage, Burrows, Nelson, Sobalvarro, & Anderson, 1997; Wang, Kelly, Kudlur, Laforetune, & Mahmke, 2008; Yu, Rodeheffer, & Chen, 2005) track program execution and report a data-race problem if the program follow a certain concurrency order. These tools produce relevant result, according to order of execution or program inputs, and can not cover all execution paths; so are not sound.

Type systems can infer and gather information about programs as well as achieving program analysis. The merits of using type systems are attesting and rationalization of properties of programs directed by their phrase structures. Type systems are actually sufficient frameworks for describing data flow analysis. A general method for producing such description was presented (Laud, Uustalu, & Vene, 2006). Type systems are used as a framework for analyzing multi-threaded programs as well as imperative programs. In (El-Zawawy, 2011), type systems were used as a framework for pointer analysis for multi-threaded programs. In (El-Zawawy & Nayel 2011) type systems were used as a framework for eliminating redundancies in multi-threaded programs.

In this paper we present a static detector. We introduce a type system that detects data-race problem for
multi-threaded programs of a simple language \textit{m-while}. We also prove the soundness the proposed type system.

The rest of this paper is organized as follows. We outline the related work in Section 2. Section 3 presents the language, a motivation example, and an operational semantics for the language. Read type system and the proof of its soundness are introduced in Section 4. In Section 5, we introduce safety type system and a proof for its soundness. Future works are outlined in Section 6.

2. Related Work

Multi-threading is a promising area of research. The most efficient challenging areas are compilation and program analysis (El-Zawawy, 2012a; Knoop & Steffen, 1999; Midkiff & Padua, 1990). The field of program analysis aims at collecting information about programs (Nielsen, Nielsen, & Hankin, 1999). Analyzing may concentrate on whole program, or focuses on each program point. There are many aspects of analyzing multi-threaded programs: pointer analysis (El-Zawawy, 2012a; El-Zawawy, 2011; El-Zawawy, 2011a; El-Zawawy, 2011b; Rugina & Rinard, 2003), optimization uses (El-Zawawy & Nayel, 2011; Knoop & Steffen, 1999; Knoop, Steffen, & Vollmer, 1996; Lee, Midkiff, & Padua, 1998; Lee, Padua, & Midkiff, 1999), data race detection (Cheng, Feng, Leiserson, Randall, & Stark, 1998; Rugina & Rinard, 2005), and deadlock detection (Blieberger, Burgstaller, & Scholz, 2000; Koskinen & Herlihy, 2008; Wang, Kelly, Kudlur, Lafortune, & Mahlke, 2008).

A data race occurs very often in multi-threaded programs. It occurs when two threads try to access the same location without proper synchronization, and one of them is write (Henzinger, Jhala, & Majumdar, 2004). The data race always causes program bugs. The output of program depends on scheduling of accessing memory. Detecting data race is a promising area of research, it has been studied extensively (Cheng, Feng, Leiserson, Randall, & Stark, 1998; Flanagan et al., 2002; Kahlon, Yang, Sankaranarayanan, & Gupta, 2007; Permandla, Roberson, & Boyapati, 2007; Young, Jhala, & Lerner, 2007). The first methodology to detect data races is the static race detection (Flanagan et al., 2002). Detectors, in this strategy, determine whether a program will ever produce a data race when run on all possible inputs. The second methodology is dynamic race detection, where potential races are detected at runtime by executing the program on a given input (Savage, Burrows, Nelson, Sobalvarro, & Anderson, 1997; Wang, Kelly, Kudlur, Lafortune, & Mahlke, 2008). In many detectors, data race and deadlock bugs are bundled together. Some static detectors, like Warlock (Sterling, January 1993), depends on the annotations formed by programmers to detect data races and deadlock problems. Using Theorem provers to detect many bugs including data race is the idea of extended static checker for Java (Flanagan et al., 2002; Leino, Saxe, & Stata, 1999). Some dynamic detectors are developed in the scientific parallel programming community (Cheng, Feng, Leiserson, Randall, & Stark, 1998; Dinning & Schonberg, 1991). Others detectors detects data race in Java-like programs (Choi et al., 2002; Praun & Gross, 2001). Eraser (Savage, Burrows, Nelson, Sobalvarro, & Anderson, 1997), as a dynamic detector, monitors programs during execution and looks for data race bugs. In general, dynamic detectors have the advantage that they can check un-annotated programs.

Type systems are known to be a good framework for analyzing programs (Laud, Uustalu, & Vene, 2006; El-Zawawy & Daoud, 2012). Type systems have been extensively used in pointer analysis for both imperative and multi-threaded programs (El-Zawawy, 2011; El-Zawawy, 2011b). Type systems have been used to detect the memory safety of multi-threaded programs (El-Zawawy, 2011; El-Zawawy, 2011b; El-Zawawy, 2011a). Type systems are used in code optimization. Partial redundancy elimination was performed via type systems for imperative programs (Saabas & Uustalu, 2009), and for multi-threaded programs (El-Zawawy & Nayel, 2011). Type systems also are used to prevent data race and dead lock in a specific Java language (Permandla, Roberson, & Boyapati, 2007).

Mathematical domains and maps between domains can be used to mathematically represent programs and data structures. This representation is called denotation semantics of programs (El-Zawawy & Jung, 2006; El-Zawawy, 2007). One of our directions for future research is to translate concepts of race detection to the side of denotation semantics. Doing so provides a good tool to mathematically study in deep race detection. Then obtained results can be translated back to the side of programs and data structures.

3. Motivation

In this section, we will present a simple example that demonstrates our motivations for this research. Firstly we define a simple language, called \textit{m-while}, that supports the multi-threading concepts. In this language, statements $s \in \text{Stm}$, arithmetic expression $a \in \text{AExp}$, and $b \in \text{BExp}$ are defined over a set of program variables $x \in \text{Var}$ in the following way:

\[
\begin{align*}
\text{l} & ::= \ x \ | \ n
\end{align*}
\]
The following segment of a program motivates our work:

\[
\begin{align*}
a & := 4; \\
b & := 6; \\
x & := a + b; \\
\text{fork} & \{ y := a + b; z := b + 5; \} , \\
& \{ a := a + 5; w := a + b; \}, \\
& \{ x := x + 4; \}, \\
\text{endfork};
\end{align*}
\]

This code shows that the variable \( a \) accessed by two threads; in the first thread with read operation \( y := a + b; \) and the other thread with write operation \( a := a + 5; \). In this case the data-race problem occurs.

### 3.1 Operational Semantics

The semantics is given in terms of states. A state is a pair, \( \sigma = (R, M) \), where \( R \) is a set of variables accessed by read operations, and \( M \) is a store. A store is a mapping from variables to integers \( M \in \text{Store} = \text{Var} \rightarrow \mathbb{Z} \). The Boolean and arithmetical expressions are interpreted as truth values and integers according to stores by the semantic function \( [\cdot]_\sigma \in \text{AExp} \cup \text{BExp} \rightarrow \text{Stores} \rightarrow \mathbb{Z} \). For arithmetic expression \( a \in \text{AExp} \), \( [a]_\sigma \) denotes the arithmetic evaluation of \( a \) in the state \( \sigma \). For Boolean expression \( b \in \text{BExp} \), \( [b]_\sigma \) denotes the truth value of \( b \) in the state \( \sigma \). We write \( \sigma \Downarrow b \) to mean that \([b]_\sigma = \text{tt}\). We note that \( \text{FV}(a) \) is the set of free variables of expression \( a \). The operational semantics of \( m\text{-while} \) are given in Figure 1. The rules show that the assignment statement actually changes the state; the free variables of the expression that has been evaluated are added to read variables, and the store assigns new value for the variable being assigned a new value. The rules for imperative statements \( \text{skip, sequence, if, and while} \) changing the pre-state as usual for their classical meaning. The last rule \( (\text{fork statement}) \), that characterizes the multi-threading, describes that the fork statement changes a state using the states of each thread. One can see that our specific description of states helps in proving the soundness of our proposed type systems.

\[
\begin{align*}
x & := a : (R, M) \rightarrow (R \uplus \text{FV}(a), M \{ x \mapsto [a]_\sigma M \}) \\
\text{skip} : (R, M) \rightarrow (R, M) & \overset{\text{skip}_\text{as}}{=} \\
s : (R, M) \rightarrow (R', M') & \overset{\text{seq}_\text{as}}{=} s_0; s_1 : (R, M) \rightarrow (R, M') \\
\sigma \Downarrow b & \overset{\text{iftrue}_\text{as}}{=} s_0 : (R, M) \rightarrow (R', M') \\
\sigma \not\Downarrow b & \overset{\text{iffalse}_\text{as}}{=} s_1 : (R, M) \rightarrow (R', M') \\
\sigma \Downarrow b & \overset{\text{while}_\text{as}}{=} s : (R, M') \rightarrow (R', M') \\
\sigma \not\Downarrow b & \overset{\text{whilef}_\text{as}}{=} s : (R, M) \rightarrow (R', M') \\
s_{\theta(i)} : (R, M_i) \rightarrow (R, M_{i+1}) & \forall i \in \{1, 2, \ldots, n\} \\
\text{fork} \{ s_1, \ldots, s_n \} & \overset{\text{fork}_\text{as}}{=} (R, M) \rightarrow (\bigcup R, M_{i+1}) \\
\theta & \text{ is a permutation on } \{1, 2, \ldots, n\}
\end{align*}
\]

Figure 1. The operational semantics of \( m\text{-while} \) language
4. Read Type System

In this section, we introduce read type system. At each program point, the read type system determines the variables that have been accessed with a read operation. This type system acts as a flag to discover the overlapping of read type system and the concurrent modified set. A program point has type $r \subseteq \text{Var}$, if all variables in $r$ are accessed by read operations (from beginning of the program to this point). The sub-typing is the set inclusion, i.e. $r \subseteq r_1$. The rules of read type system are given in Figure 2. The first rule (:= r) adds the free variables of the computed expression to the pre-type. The rules skipr, sequencer, ifr, and whiler affect pre-type as expected considering their classical meaning. The rule (conseqr) is important for weakening the pre-type and strengthen the post-type. For the rule (forkr) which characterizes the multi-threading, the post-type of fork statement is the union of all post-types of different threads.

$$
x := a : r \rightarrow r \cup FV(a)
$$

**Figure 2. The rules of read type system of m-while language**

4.1 Soundness of Read Type System

In this section, we prove the soundness of read type system. Firstly, the following definition is introduced:

**Definition 1.**

For a state $\sigma = (R, M)$, we say that $\sigma$ entails $r$, where $r \subseteq \text{Var}$ is a read type set, if $r \subseteq R$. This is written as follows $\sigma \models r \iff r \subseteq R$.

The soundness of the concerned type systems is introduced in the following theorem.

**Theorem 1.**

If $s : \sigma \rightarrow \sigma'$ and $s : r \rightarrow r'$, then $\sigma \models r \implies \sigma' \models r'$.

**Proof:**

The proof is by structural induction on rules, we demonstrate some cases:

- **Case :=r**
  
  We use the operational rule :=os. Let $\sigma \models r \implies r \subseteq R$. Hence we have $r' = r \cup FV(a) \subseteq R \cup FV(a) \subseteq R' \implies \sigma' \models r'$.

- **Case forkr**
  
  We use the operational rule forkos. From premises the following are satisfied:

  $s_{\theta(i)} : r \rightarrow r_i$, and $s_{\theta(i)} : \sigma_i : (R, M_i) \rightarrow \sigma_{i+1} : (R, M_{i+1})$. i.e.

  $\sigma_i \models r \implies \sigma_{i+1} \models r_i$ or equivalently $r \subseteq R \implies r_i \subseteq R_i$. It is enough to prove that:

  $\sigma_i \models r \implies \sigma_{i+1} \models \bigcup_i r_i$

  But, $\sigma_r \models r \implies r \subseteq R$

  $\implies r_i \subseteq R_i$

  $\implies \bigcup_i r_i \subseteq \bigcup R_i$
5. Safety Type System

In this section, we introduce the safety type system. Each program point has a type \( d \in \{true, false\} \), where true means that the program is safe at this point, and false means that the program is unsafe at this point. In the following \( C(s) \) denotes the concurrent modified set of the statement \( s \) introduced in (El-Zawawy & Nayel, 2011), and \( p_1 \land p_2 \) denotes the logical conjunction of Boolean variables \( p_1 \) and \( p_2 \). The following definition is needed:-

**Definition 2.**

The truth value of a set \( A \) is defined as follows:

\[
\text{Tr}(A) = \begin{cases} 
true & \text{if } A = \emptyset \\
false & \text{if } A \neq \emptyset
\end{cases}
\]

For any two sets \( A \) and \( B \) the following properties are satisfied

\[
\text{Tr}(A) \land \text{Tr}(B) = \text{Tr}(A \cup B)
\]

and

\[
\text{Tr}(A) \lor \text{Tr}(B) = \text{Tr}(A \cap B)
\]

The rules of safety type system are defined in Figure 3. The first rule := checks the overlapping of concurrent modified set and the read set. All other rules are straightforward. In general we conclude that, the program is safe if each program point has a type true otherwise the program is unsafe.

![Figure 3. The rules of safety type system of m-while language](image)

5.1 Soundness of Safety Type System

Firstly we define the entailment of a type \( d \) in state \( \sigma = (R, M) \) with respect to set \( A \) as follows:

\[
\sigma \models d \iff \text{Tr}(R \cap A) = d
\]

The following theorem states and proves the soundness of the safety type system.

**Theorem 2**

Let \( s : \sigma \rightarrow \sigma' \), and \( s : d \rightarrow d' \). Then \( \sigma \models_{C(s)} d \implies \sigma' \models_{C(s)} d' \)

**Proof:**

The proof is by structural induction on rules, we present some cases:

- **Case :=**
  
  Suppose \( x := a : \sigma \rightarrow \sigma' \), and \( x := a : r \rightarrow r' \), where \( \sigma = (R, M), \sigma' = (R', M') \), \( r' = r \cup \text{FV}(a) \), and \( R' = R \cup \text{FV}(a) \).
  
  From premises, \( r \subseteq R \implies r' \subseteq R' \).
  
  Let \( \sigma \models_{C(s)} d \). Then \( d = \text{Tr}(R \cap C(x := a)) \)
  
  Now \( d' = d \cup \text{Tr}(r' \cap C(x := a)) \)
  
  \[
  = \text{Tr}(R \cap C(x := a)) \land \text{Tr}(r' \cap C(x := a))
  \]
  
  \[
  = \text{Tr}(R \cap C(x := a)) \cup (r' \cap C(x := a))
  \]
  
  \[
  = \text{Tr}(R \cup r' \cap C(x := a))
  \]
= Tr((R ∪ r ∪ FV(a)) ∩ C(x := a))

= Tr((R ∪ FV(a)) ∩ C(x := a))

= Tr(R′ ∩ C(x := a))

⇒ σ′|_{C(x := a) d′}

i.e., σ|_{C(x := a) d} ⇒ σ′|_{C(x := a)} d′ which completes the proof.

- Case seq

We use the operational semantic rule seq. From premises we have:-

σ|_{C(s0)} d ⇒ σ''|_{C(s0)} d'', and σ''|_{C(s0)} d'' ⇒ σ|_{C(s0)} d'

From the definition of concurrent modified function given by (El-Zawawy & Nayel, 2011), we can conclude that for a sequence of statements s0,s1 the following issatisfied:-

C(s0) = C(s1) = C(s0,s1)

i.e., σ|_{C(s0,s1) d} ⇔ σ''|_{C(s0,s1)} d''

and

σ''|_{C(s0,s1) d''} ⇒ σ|_{C(s0,s1)} d'

then we can conclude that σ|_{C(s0,s1) d} ⇒ σ|_{C(s0,s1)} d'.

- Case fork

To prove this rule we can consider the fork statement as the following sequence of statements fork {s1}, {s1}, ..., {sn} endfork = s_{θ(1)}, s_{θ(2)}, ..., s_{θ(n)}. Now applying the sequence rule produces the proof.

5.2 Implementation

We implemented a detector based on our type system. This program checks any program of m-while language for safety. For a program of m-while language, the detector computes the modified sets of each program point and computes read sets for each program point. Then the intersection of these two sets is calculated.

6. Conclusion and Future Work

In this paper we present a static data race detector. We use type systems as a framework to implement this detector. Firstly, we present read type system which computes the variables accessed by read operations. Secondly, we present safe type system. This type system is based on read type system and decides if a program contains data race problems or not. The soundness of these type systems are discussed in this paper as well. For future work we plan to use type systems as a tool to solve more complicated problems (like deadlock, pointer dangling). We expect type systems to be an amenable and trustable framework to deal with static analyses. We also plan to improve our work in many directions including extending our language to support the object-oriented concepts.

References


